CPSC 545 Computing Systems
# Project 1: Program Graph Executor
Due: April 11th, 23:59PM

1. Goal

This project is to develop a **graphexec** program in C/C++ that will execute a graph of user programs *in parallel* using **fork**, **exec** and **wait** in a controlled fashion. Such graphs are used in compiler analysis and parallel program execution to model control- and data-dependencies. Control dependence specifies that a program cannot start until its predecessor(s) is finished. Data dependency specifies that a program requires input from its predecessor(s) before it can execute.

2. Description

Your main program **graphexec** is responsible for analyzing a graph of user programs, determining which ones are eligible to run, and running programs that are eligible to run at that moment in time. As programs in the graph finish, your program will determine which other programs in the graph have become eligible to run, execute those programs, and continue this process until all programs are finished.
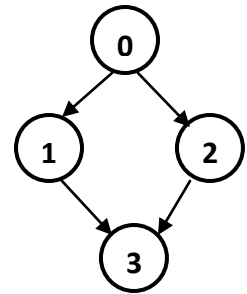
In the example graph above, node 0 can be executed first because it is not a child of any other node. After node 0 finishes executing, then nodes 1 and 2 can be executed in parallel (they both should be started immediately without waiting for either 1 or 2 to finish). Only after both 1 and 2 finish can the final node 3 be executed. Each node in the graph represents one program to be executed.

Each node will contain:
- *the program name with its arguments,*
- *"pointers" to child nodes,*
- *the input file to be used as standard input for that program, and*
- *the output file to be used as standard output for that program.*

A node becomes eligible for execution once all of its parent nodes (nodes that contain a pointer to this node) have completed their own execution. Your main program will fork and exec each of the nodes as they become eligible to run. Also, **input and output redirection** must be used so that each node can get its standard input from a file and also write its output to a file (respectively).

We will run your **graphexec** program as follows:

$ ./graphexec   some-graph-file.txt

2.1 Graph file format

In the above command, a text file will represent the structure of the graph. We have posted a sample graph file along with these specifications. Each node will be represented by a line in the following format:

program name with arguments:list of children IDs:input file:output file

[Example:] ls -l:1 2:blank-file.txt:ls-output.txt
[Example:] cat myfile1 file2:none:blank-file.txt:cat-out.txt
[Example:] gzip:none:ls-output.txt:ls-output.txt.gz

NOTE: If there are no children for a node, it *must* be specified as "none".

For convenience, the nodes will implicitly be numbered from 0 to (n-1) from the order the nodes appear in the text file (where n = total number of nodes in the graph). The children IDs will correspond to this numbering system. You may not assume that the nodes are in order from top to bottom based on their ID number. The ID numbers are used mainly to create "pointers" from parent nodes to children nodes.

2.2 Node structure

You may find helpful the node structure below:

```
enum Status { INELIGIBLE, READY, RUNNING, FINISHED };
struct Node {
        int id; // corresponds to line number in graph text file
        char prog[1024]; // prog + arguments
        char input[1024]; // filename
        char output[1024]; // filename
        int children[10]; // children IDs
        int num_children; // how many children this node has
        Status  status; // ineligible/ready/running/finished
        pid_t pid; // track it when it's running
};
```

2.3 High-level view

Your main program will first parse the graph file given in the first argument, construct a data structure that models the graph, and then start executing the nodes. Your program should determine which nodes are eligible to run, execute those nodes, wait for any node to finish, then repeat this process until all nodes have finished executing.

3. Useful system calls and functions

It is highly suggested that you make use of the following system calls or library functions:

open, fopen, fgets, fork, dup2, execvp, wait, strtok

4. Simplifying assumptions

- There will be no more than 50 nodes in any given graph.
- No node will have more than 10 children.
- The graph file will not contain extra whitespace "padding" between the colons and the values.
- The graph file will contain single spaces between the children ID numbers for that parameter.
- Each line in the graph file will not exceed 1023 characters.
- The graph will not contain cycles

5. Error handling

You are expected to check the return value of all system calls that you use in your program to check for error conditions. Also, your main program should check to make sure the proper number of arguments is used when it is executed. If your program encounters an error, a useful error message should be printed to the screen. Your program should be robust; it should try to recover from errors if possible. If the error prevents your program from functioning normally, then it should exit after printing the error message. (The use of the perror() function for printing error messages is encouraged.)

6. Documentation

You *must* include a README file which describes your program. It needs to contain the following:
- The purpose of your program
- How to use the program from the shell (syntax)
- What exactly your program does

The README file does not have to be very long, as long as it properly describes the above points. Proper in this case means that a first-time user will be able to answer the above questions without any confusion.

Within your code you should use one or two sentences to describe each function that you write. You do not need to comment every line of your code. However, you might want to comment portions of your code to increase readability.

At the top of your README file and main C source file please include the following comment:
/* CPSC545 Spring2011 Project 1
* login: cs1_login_name (login used to submit)
* Linux
* date: mm/dd/yy
* name: full_name1, full_name2 (for partner(s))
* emails: your and your partners' emails */

7. Grading [20 points]

2 pts   README file
2 pts   Makefile that works correctly to compile your program

4 pts   Documentation within code, coding, and style (indentations, readability of code, use of defined constants rather than numbers)

12 pts   Test cases (correctness, error handling, meeting the specifications, no scrambling messages on screen)

- Please make sure to pay attention to documentation and coding style. A perfectly working program will not receive full credit if it is undocumented and very difficult to read.
- The test cases will not be given to you upfront. They will be designed to test how well your program adheres to the specifications. So make sure that you read the specifications very carefully. If there is anything that is not clear to you, you should ask for a clarification.
- Make sure your code compiles and run on cs1.seattleu.edu. Failures to compile and execute you program result in zero point.
-

8. Deliverables

The following deliverables must be submitted:
- Source code files
- A README file
- A Makefile that will compile your code and produce a program called **graphexec**. Note: this Makefile will be used by us to compile your program with the make utility.

All files should be made a package by tar and submitted using the SUBMIT utility. This is your official submission that we will grade. Please note that future submissions under the same homework title OVERWRITE previous submissions; we can only grade the most recent submission. You can NOT submit your code after the deadline is passed.

9. Submission

You should connect to cs1.seattleu.edu using your account to submit your project deliverables. You can submit multiple times before the deadline. The system keeps the most recent submission. You should follow the steps to submit your project:
- Make a tar package of your deliverables (one example below)
  tar -cvf p1.tar README Makefile graphexe.h graphexe.cpp
- Submit your tar package
  /home/fac/testzhuy/CPSC545/upload   p1   p1.tar